A project report on

TITLE OF THE PROJECT REPORT

DESIGNING A MULTITHREADED WEBSERVER

Submitted in partial fulfillment for the award of the degree of

OPERATING SYSTEM – CSE2005
To Dr. Vani V.

by

NAMES OF THE Team Members with REGISTRATION No.

Nandan v. Baldha-19BCE1001
Suyash Shrivastava-19BCE1503

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Name of the school

SCOPE

Abstract

When a client sends the request, a thread is generated through which a user can communicate with the server. We need to generate multiple threads to accept multiple requests from multiple clients at the same time. The server will be able to handle multiple simultaneous service requests in parallel. In the main thread, the server listens to a fixed port. When it receives a TCP connection request, it sets up a TCP connection through another port and services the request in a separate thread

1. Objective :

The objective of the project is to implement MultiThreaded Web Server to serve

multiple incoming client request simultaneously using multiple threads.

2. Overview :

The server reads any incoming requests from the client and serves requested file from specified directory. Server have multiple threads to accomplish its' work. The number of threads can be controlled by user.

Basically, here are 3 types of threads in the web server.

1. Dispatcher/main thread
2. Scheduler thread
3. Helper thread

3. Multithreading :

In our server, one main thread is continuously listening to socket for incoming request, one scheduler thread places the incoming request in ready queue based on scheduling algorithm and helper threads (default = 10) are there to serve incoming request. Below is the brief information about each types of the threads.

3.1 Main thread / Dispatcher thread

Main thread listens continuously to socket for incoming request. As soon as new request comes to server, it parses the request, checks that it is not bad request and fills the all fields of current client structure, check file exists or not, check the type of request and puts it into the queue.

3.2 Scheduling thread

The scheduling thread comes into the play when there are multiple client requests in the queue. Scheduler thread fetch the request from the queue based on scheduling policies. The client request is served by specific algorithm. We implemented two scheduling algorithms: first is either the "First-Come, First-Served" (FCFS) and second is "Shortest Job First" (SJF).

In FCFS fetch the request in order it came and in SJF it sorts the queue based on requested file size in increasing order and fetch the shortest one.

3.3 Helper thread

When helper thread will waked up by scheduling thread, it will move from idle state to ready state serve the incoming request, serve the incoming request.

4 Implementation :

We implemented our project using C++. We used socket programming along with pthread library to implement threads as described above.

4.1 Files

Server side files:

main.h
server.h
manageRequest.h

senddata.h
main.cpp
server.cpp
manageRequest.cpp
senddata.cpp
Makefile

Client side files:

client.cpp
Input.txt

4.2 Compilation and Execution

To execute server, compile server files using `make` command and give command ./httpserver to terminal. httpserver provides below functionality when used with options.

To execute client, compile server files using `g++ client.cpp -lpthread` command and give command with options as described below.

```
Usage: ./a.out -i ipAddress -h -f inputFile

     -i followed by ip address of the server
     -f followed by input file name, default input file is
input.txt
     -h to print this help

Note : -i option is necessary
Input file format:
Number of files
<file1>
<file2>
.
.
.
<fileN>
```

Input.txt has first line as integer n and all other n lines each consisting one filename.

When it runs with -i ip address option, it sends the multiple requests to server by creating n number of threads. It reads input file and each thread will send request for one file.

Each thread sends request and receives its' data. When reply comes back on socket, each thread reads header first and then open the appropriate output file and prints filedata in it. It also shows progress percentage of received data for first two threads in order to show parallelism.

Note that HTTP header and data will be separated by CRLF (Carriage Return Line Feed).

4.3 Flow of execution

When we run our httpserver then main.cpp executes first. It parses the request given on command line and do option parsing as described in execution part.

It creates one scheduler thread and threadnum (default = 10) helper threads. So there is a scheduler thread, helper threads, a main thread.

Before creating above threads, to manage concurrency and race conditions mutex has to be used.

2 mutex variables : rqueue_lock and print_lock

2 condition variables : rqueue_cond and print_cond

Condition variables are there to avoid race condition and to awake another thread based on some condition.

```
pthread_mutex_t rqueue_lock;
pthread_cond_t rqueue_cond;
pthread_mutex_t print_lock;
pthread_cond_t print_cond;
```

rqueue_lock mutex is there to avoid concurrent use of queue(clientlist) by a scheduler thread and a main thread. print_lock mutex is there to avoid concurrent use of ready queue(requestlist) by scheduler thread and helper threads. We initialize queue_lock, print_lock, rqueue_cond, print_cond before creating threads.

A scheduler thread reference to `serveRequest_helper(void *c)` routine in manageRequest.cpp and all helper threads references to `popRequest_helper(void *c)` routine in manageRequest.cpp.

All the helper threads are detached before starting his routine. When a detached thread terminates, its resources are released immediately instead of having to wait for the its' creator thread (main thread) to join them. pthread_detach() itself mean that the thread will be detached from creator and all this thread takes responsibility of releasing resource themselves.

Then, `accept_connection()` method of server.cpp will be called. Here main thread listens to incoming client requests continuously. This function will create a socket for all the incoming client connections and make the object of client-Identity structure and fills the details about client here, pass this structure to manageRequest class.

Lets' look at below two structures : Client Identity and Client Info.

```
struct clientIdentity
{

     int acceptId;//accpetId on which connection is
established
     string ip;               // ip address of client
     int portno;              //port no of incoming request
     string requesttime;      // request time
```

```
};
struct clientInfo
{
     string r_method;          // HTTP
     string r_type;            //GET/HEAD
     string r_version;         //1.1
     string r_firstline;       //first line of GET/HEAD request
     string r_filename;        //filename
     string r_time;            //request time
     string r_servetime;       //service time
     int r_acceptid;           //acceptId from clientIdentity
     string r_ip;              //ip from clientIdentity
     u_int16_t r_portno;       //port number from
clientIdentity
     int r_filesize;           //requested files' size
     bool status_file;         //file exists or not
     string r_ctype;           //content type i.e. html,txt,jpg
     int status_code;          //like 404(NOT FOUND),200(OK)

};
```

Then `parseRequest(clientIdentity c_id)` method from `accept_connection()` is called. This function will have the current client-identity structure as its' argument. It reads the htpp request from socket sent by client, parses the request. It fills the fields of client-info using this retrieved information and client-identity information.

It prepend root directory to filename and calls `checkFilename (clientInfo c)` to check if the tilt present filename or not. It will replace tilt with its' full path accordingly and return back to parserequest().

It calls `checkRequest(clientInfo cInfo)` function that uses `fileExists(const char *filename)` function to check whether requested file is existed or not. If file does not exist, it will put filesize zero and status_file as false. If it exists then it will open that file and get the size of it and close it.
It will put this size in clientInfo filesize and set the status_file true.

Then `readyQueue (cInfo)` is called. It will put the that clientinfo object into clientlist queue. While putting this into clientlist, it locks rqueue_lock to make clientlist thread safe. It will also send `pthread_cond_signal(&rqueue_cond)` to awake the rqueue_cond variable if it is waiting on it, unlocks rqueue_lock.

`pthread_cond_wait (&cond_variable, &mutex)` unlocks mutex variable, puts cond_variable on wait. It can be awaken by pthread_cond_signal (&mutex). After waking, it locks mutex and resumes its' execution.

Recall that main.cpp creates scheduler thread and all helper threads initially. Scheduler thread references to `popRequest_helper (void *c)` routine. That redirects to `popRequest()` function. This function has infinite loop and based on selected

scheduling policy(either FCFS or SJF) it will pop the request from clientlist and push it into requestlist. When it pops request from clientlist it uses rqueue_lock mutex for locking and unlocking of the clientlist. But initially clientlist is empty in which case scheduler thread will wait until clientlist gets any entry. To handle this situation, it uses `pthread_cond_wait (&rqueue_cond, &rqueue_lock)` to wait on rqueue_cond variable, unlocks rqueue_lock mutex. Thread will wait at this point until wake up signal arrives.

Whenever `pthread_cond_signal (&rqueue_cond)` sends signal, it will awake this rqueue_cond variable and this thread resumes its' execution by locking rqueue_lock mutex. Note that this signal will be sent by `readyQueue(clientInfo cInfo)` whenever it adds entry into clientlist. Thus schedular thread is waiting on rqueue_cond variable then it will be awaken by signal to let it schedule the request in clientlist.

Similarly helper thread do the same thing along with helper thread. When scheduler thread pushes the request into requestlist, it will use print_lock mutex to make use of requestlist thread safe. It will also send the `pthread_cond_signal (&print_cond)` to awake thread if it is waiting on print_cond variable.

Again recall that main.cpp creates helper threads and these thread references to `serveRequest_helper (void *c)` routine. In this routine `serveRequest ()` function called. In this function continuously helper threads are acting to serve the client request.

In this function there is infinite loop and thread is taking one request from requestlist queue and giving request to `sendData (clientInfo c)` function by putting serve time in clientinfo . print_lock mutex is used so that only one thread access requestlist at particular time. When thread initially created that time requestlist will be empty so at that time thread waits on condition variable print_cond and unlock print_lock mutex so that scheduler thread can use that mutex. When `pthread_cond_signal (&print_cond)` will come it awakes print_cond variable and thread resumes its' work. Whenever scheduler puts request into requestlist it sends
`pthread_cond_signal (&print_cond)` to wake up thread if it is waiting on print_lock cond variable.

The `sendData (clientInfo c)` function of sendata class is called. It will note down the starting time of service, last modified time of requested file, size of the requested file and type(txt, html, etc.) of requested file. Using these information and having other information such that method, content type , header is built. If the request type is "HEAD" then our work is done and header will be sent else if the request type is "GET" then it will send the file after header.

If -l command is given then request need to logged in that given logfile, for that `generatingLog (clientInfo c)` function is called. It will open the log file, make a entry in file using clientInfo(ip address, request time , serve time, request, status code, filesize) and close the logfile.

If consoleLog boolean variable is set then the log(ip address, request time , serve

time, request, status code, filesize) will be displayed on the console by `displaylog (clientInfo c) function`.

If file doesn't exist then it will call `listingDir()` to list down the existing directories. `listingDir()` uses `sortDirectory()` function to sort list of directory. After that, `sendData()` function will close that client connection using clients' acceptId.

Conclusion

Less time is spent outside the accept() call.
Long running client requests do not block the whole server
As mentioned earlier the more time the thread calling serverSocket.accept() spends inside this method call, the more responsive the server will be. Only when the listening thread is inside the accept() call can clients connect to the server. Otherwise the clients just get an error.

In a singlethreaded server long running requests may make the server unresponsive for a long period. This is not true for a multithreaded server, unless the long-running request takes up all CPU time time and/or network bandwidth.

Future work

The future expansion of this project is not very vast. We can introduce more new threads to increase the efficiency and to handle more number of clients at the same time. Multiple Handling thread can be used to decrease load on each separate threads.

References

https://www.geeksforgeeks.org/socket-programming-cc/
https://www.educative.io/blog/modern-multithreading-and-concurrency-in-cpp
https://codereview.stackexchange.com/questions/143286/multithreaded-client-server-communication
https://www.geeksforgeeks.org/socket-programming-in-cc-handling-multiple-

clients-on-server-without-multi-threading/

Appendices

Sample Code

Client.cpp

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <bits/stdc++.h>
#include <arpa/inet.h>
#include <pthread.h>

#define MAXLINE 256
#define MAXFILES 30
#define MAXSUB 10
#define MAXRESPONSE 10
using namespace std;
pthread_t *threads;
string ipaddress, exefile, input;
int noFiles;
string files[MAXFILES];
int data[MAXFILES];
long received[MAXFILES], total[MAXFILES];
void
printhelp (char *outputfile)
{
      printf ("\nUsage: %s -i ipAddress -h -f inputFile \n\n",
outputfile);
      printf ("\t-i followed by ip address of the server\n");
      printf ("\t-f followed by input file name, default input file
is %s\n", input.c_str ());
      printf ("\t-h to print this help\n\n");
```

```
	printf ("Note : -i option is necessary\n");
	printf ("Input file format:\n");
	printf ("Number of
files\n<file1>\n<file2>\n.\n.\n.\n<fileN>\n\n");
}
void *
clientRequest (void *xx)
{
	int sockfd = 0, n = 0;
	struct sockaddr_in serv_addr;

	if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
	{
		printf ("\n Error : Could notd create socket \n");
		return NULL;
	}

	memset (&serv_addr, '0', sizeof(serv_addr));

	serv_addr.sin_family = AF_INET;
	serv_addr.sin_port = htons(8080);

	if (inet_pton (AF_INET, ipaddress.c_str (),
&serv_addr.sin_addr) <= 0)
	{
		printf ("\n inet_pton error occured\n");
		return NULL;
	}

	if (connect (sockfd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr)) < 0)
	{
		printf ("\n Error : Connect Failed \n");
		return NULL;
	}
	ofstream file;
	int y = *((int *) xx);
	// Form request
	char sendline[MAXLINE+1], recvline[MAXLINE+1];
	sprintf (sendline, "GET %s HTTP/1.1\r\nHost: %s\r\n\r\n",
files[y].c_str (), ipaddress.c_str ());

	file.open (files[y].c_str (), ios::ate);
	if (write (sockfd, sendline, strlen (sendline)) >= 0)
	{
		//Read the header
		if ((n = read (sockfd, recvline, MAXLINE)) > 0)
			recvline[n] = '\0';
```

```
            string header (recvline);
            string temp = "Content-Length:";
            string crlf = "\r\n\r\n";
            int pos = header.find (temp);
            int pos2 = header.find (crlf);
            total[y] = atoll (header.substr (pos + temp.length (),
pos2 - pos).c_str ());
            if (pos2 + crlf.length () <= MAXLINE - 1)
            {
                  received[y] += header.length () - pos2 - crlf.length
();
                  // Showing progress of only first two files on th
same line console in order to show parallelism.
                  if (noFiles >= 2)
                  {
                        int percent = (100.00 * received[0]) /
total[0];
                        int percent2 = (100.00 * received[1]) /
total[1];
                        printf ("\r%s : %lld/%lld  %d %%  , %s :
%lld/%lld  %d %%", files[0].c_str (), received[0], total[0],
percent, files[1].c_str (), received[1], total[1], percent2);
                  }
                  file << header.substr (pos2 + crlf.length (),
header.length () - pos2 - crlf.length () + 1);
            }
            // Read the response
            while ((n = read (sockfd, recvline, MAXLINE)) > 0)
            {
                  received[y] += n;
                  // Showing progress of only first two files on th
same line console in order to show parallelism.
                  if (noFiles >= 2)
                  {
                        int percent = (100.00 * received[0]) /
total[0];
                        int percent2 = (100.00 * received[1]) /
total[1];
                        printf ("\r%s : %lld/%lld  %d %%  , %s :
%lld/%lld  %d %%", files[0].c_str (), received[0], total[0],
percent, files[1].c_str (), received[1], total[1], percent2);
                  }
                  recvline[n] = '\0';
                  file << recvline;
            }
      }
      file.close ();
      return NULL;
```

```
}
int
main (int argc, char *argv[])
{
	int opt = 0;
	input = "input.txt";
	bool flagIp = false;
	while ((opt = getopt (argc, argv, "i:f:h")) != -1)
	{
		switch (opt)
		{
			case 'i':
				ipaddress = optarg;
				flagIp = true;
				break;
			case 'f':
				input = optarg;
				break;
			case 'h':
				printhelp (argv[0]);
				return 0;
			default:
				return 0;
		}
	}
	if (flagIp == false)
	{
		printf ("Please provide -i <ip-address> \nUse -h option
for help\n");
		return 0;
	}
	ifstream f1;
	f1.open (input.c_str());
	if (!f1.is_open ())
	{
		printf ("Error opening file\n");
		return 0;
	}
	char temp[256];
	f1.getline (temp, 256);
	noFiles = atoi (temp);
	for (int i = 0; i < noFiles; i++)
	{
		f1.getline (temp, 256);
		files[i].assign (temp);
		if (*files[i].rbegin () == '\r')
			files[i].erase (files[i].length () - 1);
		data[i] = i;
```

```
	}
	threads = new pthread_t[noFiles];
	if (noFiles >= 2)
		printf ("Showing progress of only first two files\n");
	for (int i = 0; i < noFiles; i++)
		pthread_create (&threads[i], NULL, clientRequest, (void
*) &data[i]);
	for (int i = 0; i < noFiles; i++)
		pthread_join (threads[i], NULL);
	return 0;
}
```

Sever.cpp

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include "server.h"
#include "manageRequest.h"
#include "main.h"
#include <unistd.h>
#include <cstdlib>
using namespace std;

// This function will create the socket for all the incoming client
connections & update the client structure and pass
// this structure to manageRequest class
void
RunServer::accept_connection ()
{
	if ((sockId = socket (AF_INET, SOCK_STREAM, 0)) < 0)
	{
		perror ("Socket");
		exit (1);
	}

	/*---Initialize address/port structure---*/
	bzero (&self, sizeof(self));
	self.sin_family = AF_INET;
	self.sin_port = htons(port);
	self.sin_addr.s_addr = INADDR_ANY;
	int option_value = 1;
	if (setsockopt (sockId, SOL_SOCKET, SO_REUSEADDR,
&option_value, sizeof(int)) == -1)
```

```
	{
		perror ("setsockopt");
		exit (1);
	}
	/*---Assign a port number to the socket---*/
	if (bind (sockId, (struct sockaddr*) &self, sizeof(self)) !=
0)
	{
		perror ("socket--bind");
		exit (1);
	}

	/*---Make it a "listening socket"---*/
	if (listen (sockId, 20) != 0)
	{
		perror ("socket--listen");
		exit (1);
	}

	while (true)
	{		// Main thread will listen continuously
		int acceptId;
		struct sockaddr_in client_addr;
		int addrlen = sizeof(client_addr);

		/*---accept a connection (creating a data pipe)---*/
		acceptId = accept (sockId, (struct sockaddr*)
&client_addr, (socklen_t *) &addrlen);
		time_t tim = time (NULL);
		tm *now = localtime (&tim);
		char currtime[50];
		//cout<<"here\n"<<clientport<<"here\n";
		if (strftime (currtime, 50, "%x:%X", now) == 0)
			perror ("Date Error");
		string requesttime (currtime);
		clientIdentity cid;
		cid.acceptId = acceptId;
		cid.ip = inet_ntoa (client_addr.sin_addr);
		cid.portno = ntohs(client_addr.sin_port);
		cid.requesttime = requesttime;
		cout << cid.ip << " " << cid.portno << " " <<
cid.requesttime << endl;
		M->parseRequest (cid);
	}
}
```

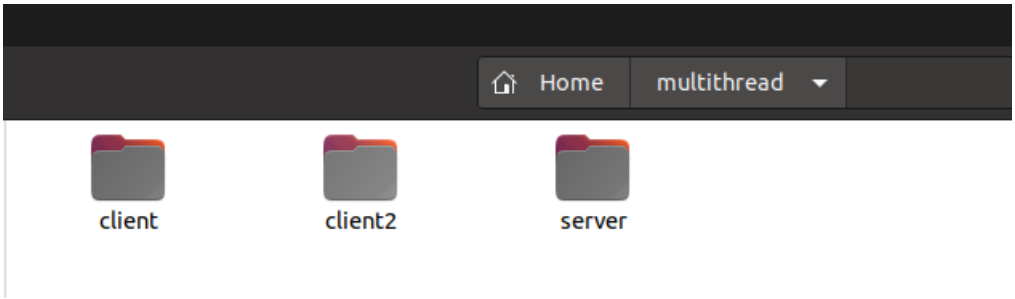
Main.cpp

```
#include "main.h"
```

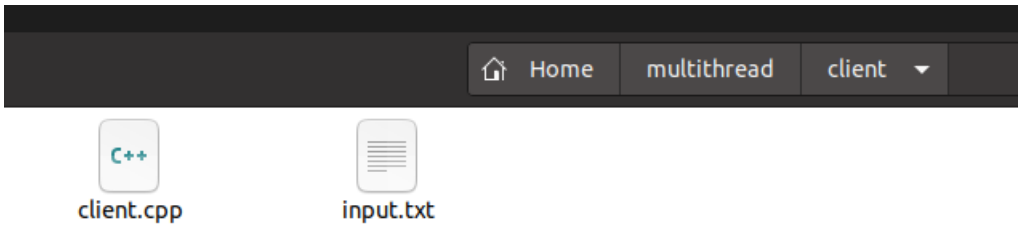
```
#include "server.h"
#include "manageRequest.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>
#include <string>
#include <cstring>
#include <unistd.h>

using namespace std;
#define MAXTHREADS 30

int port = 8080;
bool consoleLog = false;
bool logging = false;
string l_file = "log.txt";
string scheduling = "FCFS";
int threadnum = 10;
string rootdir = "./resources/";

int sockId = 0;
ManageRequest *M = new ManageRequest ();
pthread_mutex_t rqueue_lock;
pthread_cond_t rqueue_cond;
pthread_mutex_t print_lock;
pthread_cond_t print_cond;
pthread_t thread_scheduler;
pthread_t threads[MAXTHREADS];
RunServer *run = new RunServer ();



void
sigint_handler (int signum)
{
      close (sockId);
      delete M;
      M = NULL;
      delete run;
      run = NULL;
      pthread_mutex_destroy (&rqueue_lock);
      pthread_cond_destroy (&rqueue_cond);
      pthread_mutex_destroy (&print_lock);
      pthread_cond_destroy (&print_cond);
      pthread_cancel (thread_scheduler);
      for (int i = 0; i < threadnum; i++)
```

```
            pthread_cancel (threads[i]);
      exit (signum);
}

int
main (int argc, char *argv[])
{

      signal (SIGINT, sigint_handler);

      pthread_mutex_init (&rqueue_lock, NULL);
      pthread_mutex_init (&print_lock, NULL);
      pthread_cond_init (&rqueue_cond, NULL);
      pthread_cond_init (&print_cond, NULL);
      pthread_create (&thread_scheduler, NULL,
&ManageRequest::popRequest_helper, M);
      for (int i = 0; i < threadnum; i++)
            pthread_create (&threads[i], NULL,
&ManageRequest::serveRequest_helper, M);
      run->accept_connection ();

      return 0;
}
```

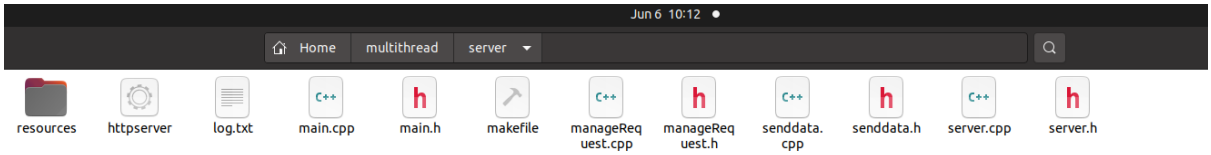
Screen shots



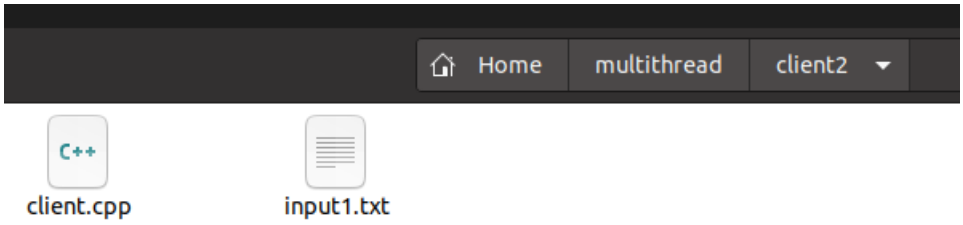
Multithread folder



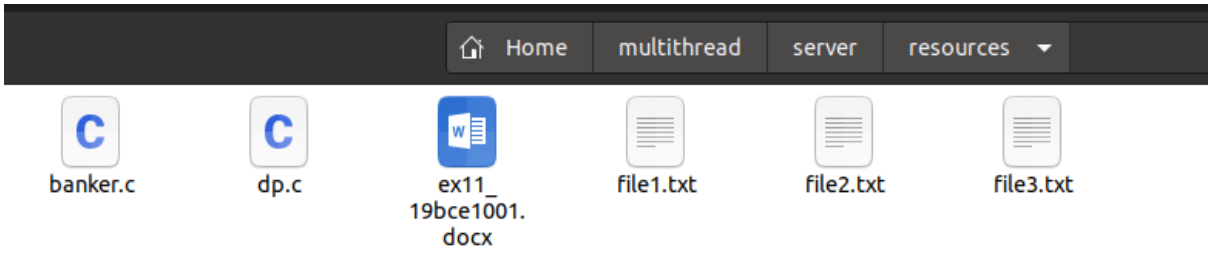
Client folder



Server Folder



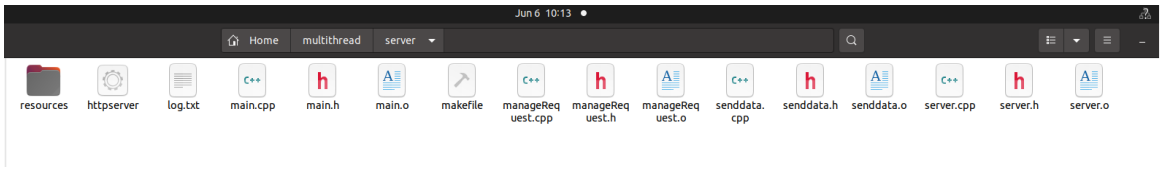
Client2 Folder



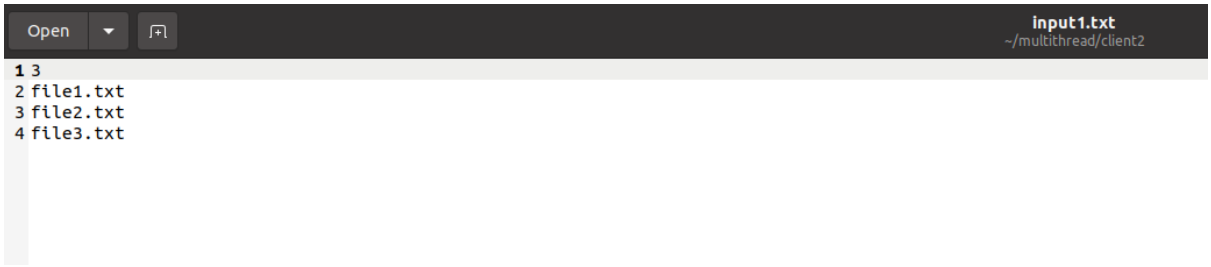
Resources Folder

```
nandan@nandan-VirtualBox:~/multithread/server$ make
g++     -c -o main.o main.cpp
g++     -c -o server.o server.cpp
g++     -c -o senddata.o senddata.cpp
g++     -c -o manageRequest.o manageRequest.cpp
g++ -o httpserver main.o server.o senddata.o manageRequest.o -lpthread
nandan@nandan-VirtualBox:~/multithread/server$
```

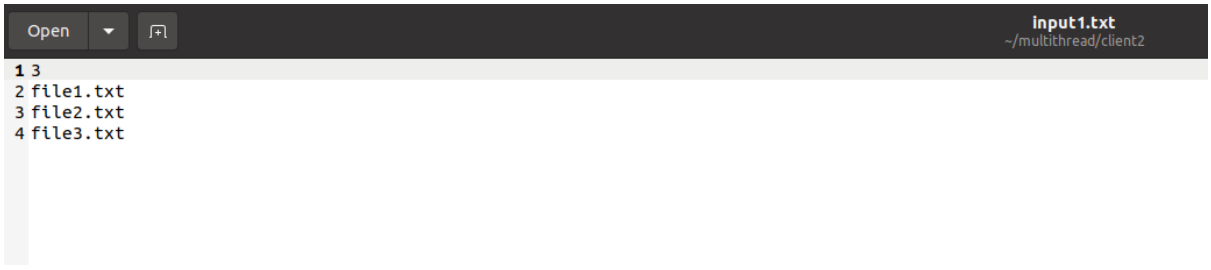
Run make command in server



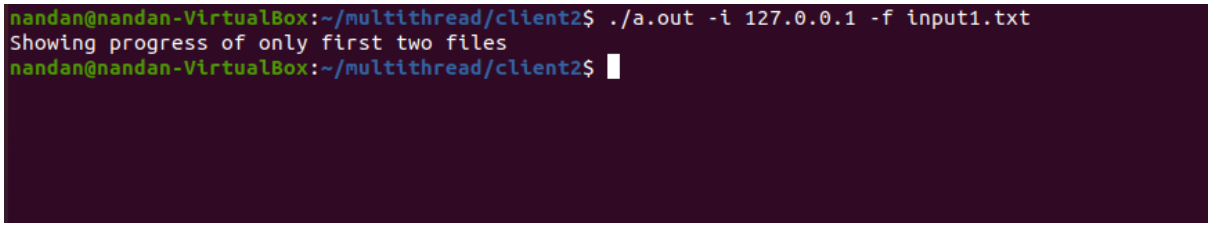
Object file crated in server



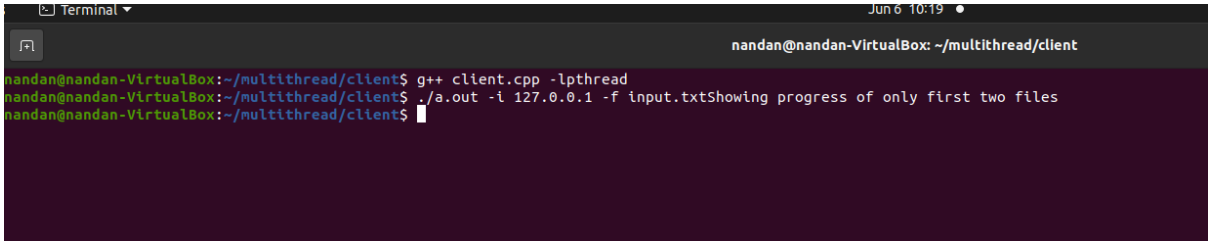
Input.txt file



Input1.txt File



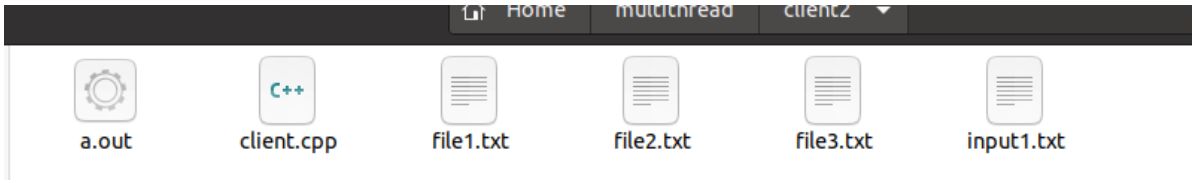
Request from client 2



Request from client 1

```
nandan@nandan-VirtualBox:~/multithread/server$ ./httpserver
127.0.0.1 49412 06/06/21:10:16:56
127.0.0.1 49416 06/06/21:10:16:56
Serving request ./resources/file1.txt
127.0.0.1 49418 06/06/21:10:16:56
127.0.0.1 49414 06/06/21:10:16:56
127.0.0.1 49420 06/06/21:10:16:56
Serving request ./resources/dp.c
Serving request ./resources/file3.txt
Serving request ./resources/ex11_19bce1001.docx
Served request of ./resources/file1.txt
Serving request ./resources/banker.c
Served request of ./resources/dp.c
Served request of ./resources/file3.txt
127.0.0.1  [06/06/21:10:16:56]  [06/06/21:10:16:56]  GET file1.txt HTTP/1.1 200 85
127.0.0.1  [06/06/21:10:16:56]  [06/06/21:10:16:56]  GET dp.c HTTP/1.1 200 1403
Served request of ./resources/banker.c
127.0.0.1  [06/06/21:10:16:56]  [06/06/21:10:16:56]  GET banker.c HTTP/1.1 200 2811
127.0.0.1  [06/06/21:10:16:56]  [06/06/21:10:16:56]  GET file3.txt HTTP/1.1 200 32
Served request of ./resources/ex11_19bce1001.docx
127.0.0.1  [06/06/21:10:16:56]  [06/06/21:10:16:56]  GET ex11_19bce1001.docx HTTP/1.1 200 662981
127.0.0.1 49422 06/06/21:10:17:06
127.0.0.1 49424 06/06/21:10:17:06
Serving request ./resources/file1.txt
Served request of ./resources/file1.txt
127.0.0.1  [06/06/21:10:17:06]  [06/06/21:10:17:06]  GET file1.txt HTTP/1.1 200 85
127.0.0.1 49426 06/06/21:10:17:06
Serving request ./resources/file2.txt
Serving request ./resources/file3.txt
Served request of ./resources/file3.txt
Served request of ./resources/file2.txt
127.0.0.1  [06/06/21:10:17:06]  [06/06/21:10:17:06]  GET file2.txt HTTP/1.1 200 27
127.0.0.1  [06/06/21:10:17:06]  [06/06/21:10:17:06]  GET file3.txt HTTP/1.1 200 32
```

Requests on server



Client 2 get file1.txt,file2.txt and file3.txt

Client 1 get banker.c,dp.c,ex11_19bce1001.docx,file1.txt and file3.txt